
CASE Tool Editor
Dashboard Documentation
02162 Software Engineering 2 - Group 3
IMM, Technical University of Denmark

December 21, 2007

David Volquartz Lebech (s052275)

Andreas Schmidt Jensen (s052271)

Michael Thomassen (s052327)

Mikael Andersen (s973748)

Kristian Andersen (s052233)

Michael David Anyaogu (s052783)

Rafael Olmos Galve (s071150)



Contents

1	Introduction(s973748,s052783)	3
1.1	Product Description	3
1.2	Overview	3
1.3	Purpose	4
2	Requirements Specification(Everybody)	5
2.1	Overall Description	5
2.1.1	Product Perspective	5
2.1.2	Concepts	5
2.1.3	User classes and characteristics	6
2.1.4	End user	6
2.1.5	Engineer	6
2.1.6	Operating environment	6
2.1.7	Design and implementation constraints	7
2.1.8	User documentation	7
2.1.9	Assumptions and dependencies	7
2.2	UML Class Diagrams	8
2.2.1	Component Definition	8
2.2.2	Deployment	9
2.2.3	Simulation	11
2.2.4	Dashboard	12
2.2.5	Sub-model relations	13
2.3	Use Cases	15
2.3.1	Component Editor	15
2.3.2	Deployment Editor	17
2.3.3	Simulation	19
2.3.4	Dashboard Editor	21
2.3.5	Dashboard	23
3	System Features for Simulator (s052271)	26
3.1	Revised Simulation algorithm	26
4	System Features for Dashboard (s973748, s052233)	27
4.1	Create new Dashboard Project	27
4.2	Load Dashboard Project	27
4.3	Save Dashboard Project	27
4.4	Visual Element Setup	27
4.5	Visual Element Editing	27
4.6	Sensors	27
4.6.1	Description	27
4.6.2	Stimulus/Response sequences	28
4.7	Actuators	28

4.7.1	Description	28
4.7.2	Stimulus/Response sequences	28
4.8	Controllers	28
4.8.1	Description	28
4.8.2	Stimulus/Response sequences	29
5	External Interface Requirement (s973748, s052271, s052327)	30
5.1	User Interfaces	30
5.2	Hardware Interfaces	30
5.3	Software Interfaces	30
5.3.1	Simulator	30
5.3.2	Dashboard Editor	30
6	Architecture(s052275)	31
6.1	Model	32
6.2	View	32
6.3	Control	32
7	Implementation and test(s052271,s052275,s052327)	34
7.1	Editor	34
7.1.1	Autogeneration of code	34
7.1.2	Implementation issues	34
7.1.3	Test	38
7.2	Simulator	38
7.2.1	Simulation	38
7.2.2	Algorithm	39
7.2.3	Test	43
8	User Interface(s071150, s052233)	48
8.1	Dashboard Editor	48
8.1.1	Loading a deployment into the Dashboard Editor	48
8.1.2	The main window(1)	48
8.1.3	The palette(2)	49
8.1.4	Properties view(3)	49
8.1.5	Simulation View	52
9	Discussion and conclusion (s052275)	54
9.1	Problems encountered	54
9.2	Future releases	55
9.3	Conclusion	55

1 Introduction(s973748,s052783)

This document includes requirements as well as system specifications for a project created by Group 3 in 02162 Software Engineering 2 Fall 2007. The final software will serve as a tool for Computer Aided Software Engineering (CASE) and will be developed by all participants in 02162.

1.1 Product Description

Computer Aided Software Engineering or CASE refers to software which facilitates software development and modeling. This project focuses on applications of CASE with regard to embedded systems. Specifically modeling, simulating and visualizing such systems on a component basis. The system does not provide tools for actual embedded code generation or development. Instead it seeks to aid the phases of modeling and design which would precede actual implementation.

For further insights into the actual functionality of the system please refer to section 2.

1.2 Overview

The software which is being developed can be divided into five different parts as follows:

1. Component Definition Editor
2. Deployment Editor
3. Simulator
4. Dashboard Viewer and Dashboard Editor
5. Interaction Diagram Editor and Recorder

The software is being implemented by four teams each with an editor they are responsible for. In addition to the editor, every team is required to implement the Simulator.

The Component Definition Editor, Deployment Editor and Simulator represent the core of the project and every team is therefore required to produce requirement specifications for these three parts besides the editor they are responsible for.

This report is written by Group 3, and will therefore emphasize the Simulator and Dashboard related sub systems. By the end of the semester each group must present a finished sub system which will then be integrated with the other sub systems to create a fully functional tool.

1.3 Purpose

This document contains requirements specification for the Simulator and Dashboard Editor and Viewer.

Additionally the document includes systems specifications for the Simulator and Dashboard Editor and Viewer as well as a description of the user interface.

It is intended to be used by members of the project team (Group 3) during design, development and testing of the system.

2 Requirements Specification(Everybody)

2.1 Overall Description

2.1.1 Product Perspective

The Dashboard is part of a larger system (CASE tool). This system allows modeling, analyzing, simulating and visualizing a model of an embedded system. It is not easy to know if an embedded system is actually working before it has been tested.

That's why we also have to develop a simulator. Without a simulator, the engineer would have to build his system using real hardware components every time he wanted to test it. This is both expensive and time consuming. Using the internal logic of the different components, the Simulator is able to show the exact same behavior as the real hardware system would do. Furthermore, the Simulator stores all runtime information during the simulation, which can be used later for debugging by the engineer.

The Dashboard Editor is responsible for providing functionality which allows an engineer to set up a visual environment used for system analysis during simulation.

The Dashboard Viewer is the part of the CASE Tool which gives the user (see subsection 2.1.3 for more information on the different users of the system) a feeling for the system. For this, a graphical representation is needed. The Dashboard Viewer is used by the engineer to help him analyze the system and in this way identify problems. He needs a more detailed view than the end user. In the following subsections, we will go into details with each part of the Dashboard Viewer.

In the final product the Dashboard Editor and the Dashboard Viewer has been merged together to a single part, referred to as the Dashboard or the Dashboard Editor

2.1.2 Concepts

An embedded system created in the CASE Tool consists of three types of components: Actuators, sensors and controllers.

The *actuators* interact with the world. This could for example be a wiper on a car that wipes the rain of the windows.

The *sensors* are able to sense something. In the car example, this could be a rain sensor sensing whether it is raining. Sensors are able to output several different states depending on their input. Sensors might output one state when it's not raining, another state when it's raining a little and a third state when it's raining a lot.

Controllers (also known as software components) are used to simulate logic circuits and software and function like a simple state machine. A controller will change state dependant on input and output messages whenever

it switches state. Controllers are typically used to translate input data to output data according to a series of logic operations.

Communication between components is handled by *messages* which are simple data read or written by components. Messages can be transferred between components through busses or in some cases directly as described in the following.

Components are organized into groups called *nodes*. Each component is assigned to a node. If two components are connected to the same node, they are able to communicate without using a bus. To communicate with components on a different node, they will need to communicate through a bus.

2.1.3 User classes and characteristics

The system will be used by different kinds of users. Each user will have different background, education, interests and expectations. It is important to keep this in mind when designing the system. In this subsection we will go into details with the different users.

2.1.4 End user

This is the person who will be buying or using the embedded system. He will thus be using the Dashboard to get a feeling for the system. He will normally not have any significant computer skills, others than using basic functions of a computer, and should therefore be presented by a system, that easily provides exactly the level of details he need. He will be able to see a graphical representation of each component (sensors and actuators). The actual calculations behind is hidden from the end user, since his interests is *what*, not *how*.

The end user has the possibility to change states via the user interface.

2.1.5 Engineer

The engineer (alone or as part of a team) is the embedded system designer and is therefore responsible for all technical aspects of a simulation and corresponding visualization. The engineer will require a detailed look at things and should be able to see exactly what happens in a simulation if desired. This will allow the engineer to debug systems and identify potential problems and errors in general.

2.1.6 Operating environment

OE-1 The Simulator, Dashboard Editor and Dashboard must be able to run in Eclipse as a plug-in of the type editor.

OE-2 The Simulator, Dashboard Editor and Dashboard must work in every operating system, that Eclipse supports.

2.1.7 Design and implementation constraints

CO-1 The final system must run under the Eclipse v. 3.3 (Europa) with plugins for EMF, GEF and GMF installed.

CO-2 All program code will be written in the Java language using Java Standard Edition 6 (Java SE 6).

CO-3 All non-trivial implementation details must be well commented.

CO-4 The Dashboard shall conform with the provided model for the entire system (including component and deployment editors).

CO-5 The Dashboard will be implemented using the Eclipse technologies EMF, GEF and GMF.

2.1.8 User documentation

UD-1 The implementation will be documented where as it is deemed necessary.

UD-2 The final report of group 3 will contain a detailed user manual. The documentation must describe all major functionality of the editor and provide detailed examples.

2.1.9 Assumptions and dependencies

AS-1 We assume that every part of the CASE Tool is implemented according to the agreed model.

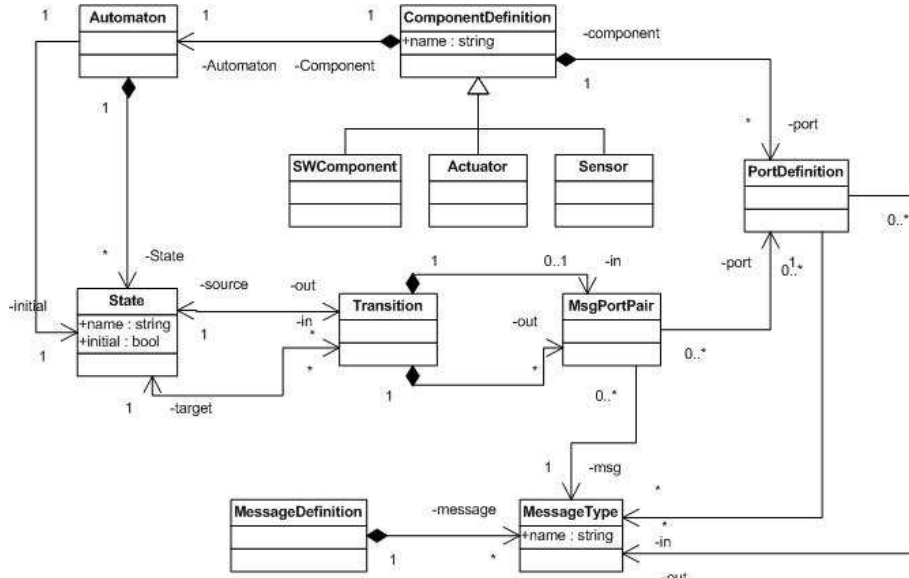


Figure 1: Class diagram for component definition

2.2 UML Class Diagrams

In this subsection we provide and describe UML class diagrams of each part of the system. The following subsections each presents a subsystem accompanied by both a general description and descriptions for individual objects. Finally an overview of the model as whole is provided.

2.2.1 Component Definition

Component Definition The component definition object consists of an automaton (state machine) and a series of port definitions. The automaton defines the components behavior when stimulated by input received via ports. The component may be instantiated as three different classes: The **SWComponent** which models a logic circuit or software program, the **Actuator** which models a device which interacts with the environment (i.e. a motor,) and finally the **sensor** which monitors changes in the environment. Internally all components are modeled as an automaton.

Automaton The automaton is a state machine. The machine starts out in an initial state and potentially changes state when its hosting component receives a message. The combination of message type, receiving port and current state determines what state (if any) the automaton will change to.

PortDefinition A part of every component is the Ports. Ports may be connected to other components directly or by means of a bus. Ports make

enables components to communicate with each other. A component can have arbitrarily many ports but one port can only be associated with one component. A port has a list of both incoming and outgoing messages associated with the port.

MessageDefinition Messages are what ports send and receive and what components use for communication. Messages by themselves have little meaning, but in combination with ports (and transitions) they help drive the system states. Messages are associated with the automaton through message/port pairs referenced by transitions.

MsgPortPair The MsgPortPair represents a pairing between a message type and a port. Such combinations may be referenced by transitions and thereby trigger state changes in the automaton (see subsection 2.2.1).

State and Transition A Transition has a source and target state, it also has a trigger message pair and one or more outgoing message pairs. A State may have one or more transitions. If a message pair is registered by the state machine it checks if this message pair is a trigger for a transition which has the current state as its source. If the message pair is a trigger for one of the transitions the state machines current state is changed to the transitions target state. The transitions outgoing message pairs are then sent to their respective ports. The State Machine consists of a finite number of states. The states are connected by transitions. When a message is received by a port the state machine checks if this is to trigger a state transition. If so the state machine changes its current state, if not the state remains the same. When the state machine is launched it always starts in a predefined initial state.

2.2.2 Deployment

Deployment The Deployment model is the part of the CASE tool which defines how created components relate and communicate with each other. Figure 2 shows the metamodel for the deployment diagram. Below there is supplementary information on the diagram.

ComponentInstance The ComponentInstance is an instance of a component as described in subsection 2.2.1. The deployment model divides components into hardware (HWComponentInstance) and software (SWComponentInstance) instances. Hardware corresponds to components of types actuator and sensor while software corresponds the SWComponent definition. The main difference between the two classes of components is the way they interact with other components. Hardware components must be connected

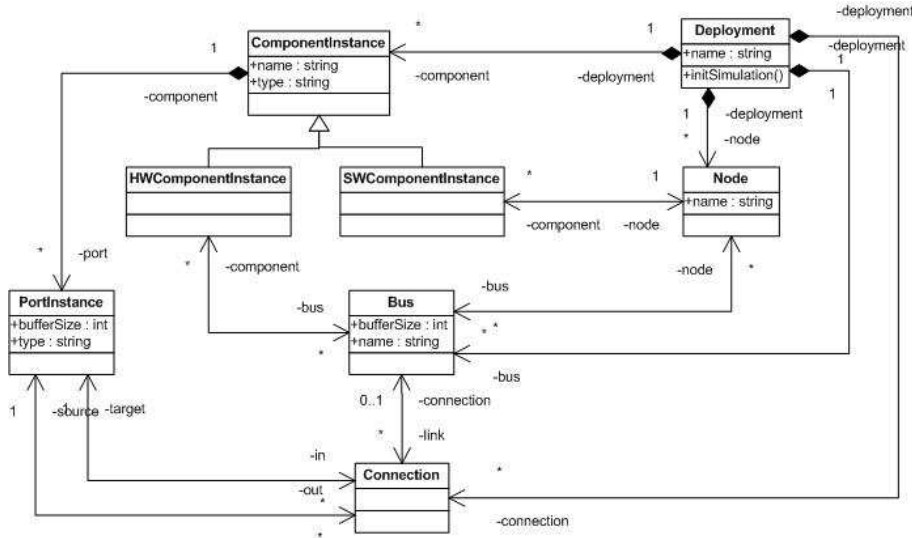


Figure 2: Class diagram for deployment

through a bus to communicate while software components may communicate with components in the same node (See subsection 2.2.2) as themselves.

Node Nodes represent processing units. A node may contain several instances of software components. Components within the same node may communicate without busses.

PortInstance This class is an instance of the PortDefinition. Ports may connect to the rest of the system through Connections and ultimately busses.

Connection A Connection specifies which two components communicate. By specifying a port on each of the two components, it is possible to establish a link between them. Each port must be able to understand all the different messages, the opposite port can send. Valid messages for a port are defined in the PortDefinition.

Bus and Connection Busses are used for communication between nodes, sensors and actuators. In other words, it transfers messages between components which are not directly connected. The bus references a number of connections, and it can therefore only tell us **who** are communicating, not **how** they are communicating. It has a limited transfer capacity of one message at a time. If the bus receives multiple messages at the same time, it will queue the messages in a buffer and transmit them one at a time.

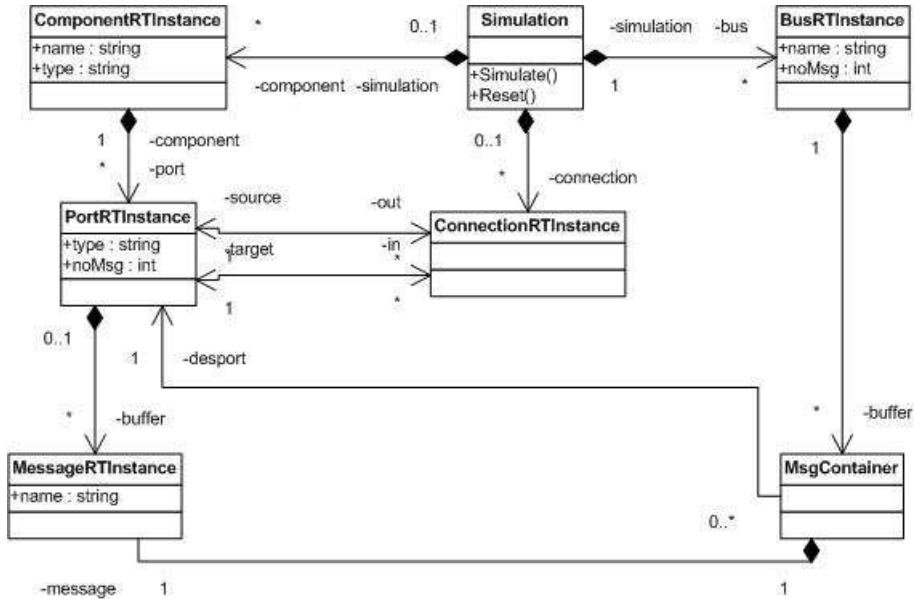


Figure 3: Class diagram for simulation

2.2.3 Simulation

The Simulator runtime is composed of a series of classes each corresponding to components or busses in the Deployment model.

ComponentRTInstance This object corresponds to runtime instances of actuators, sensors or Software components in the Deployment model.

BusRTInstance The bus runtime corresponds to runtime instances of classes of the type Bus in the deployment.

PortRTInstance The port runtime corresponds to instances of classes of the type PortInstance in the deployment.

ConnectionRTInstance The connection runtime corresponds to instances of classes of the type Connection in the deployment.

MessageRTInstance The message runtime corresponds to instances of classes of the type MessageType in the Component definition.

MsgContainer The MsgContainer object maintains the relationship between messages and ports while in the bus.

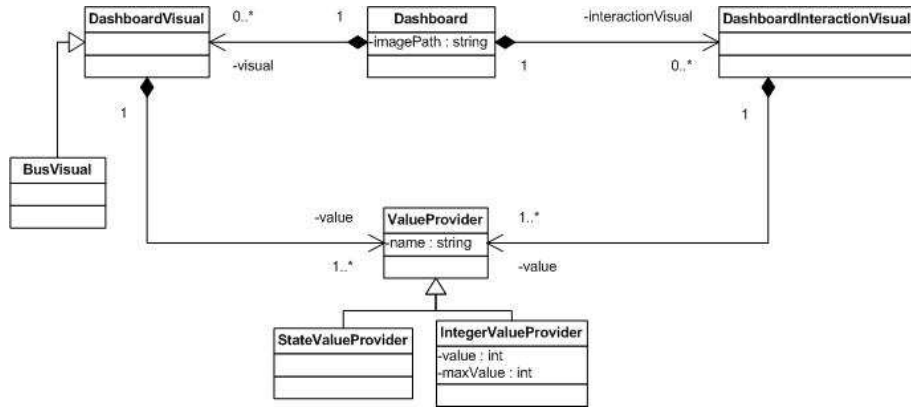


Figure 4: Class diagram for dashboard

2.2.4 Dashboard

Dashboard The Dashboard metamodel consists of metaobjects each corresponding to objects in a deployment model. Each object is responsible for visualizing the state of its respective counterpart in the deployment model. The dashboard model is outlined in figure 4. The dashboard is composed of only two different types of classes: A *DashboardInteractionVisual* and a *DashboardVisual*.

DashboardVisual The DashboardVisual is a basic visual component which represents instances of Actuators and Sensors - hardwarecomponents in other words. Such visuals are represented by images corresponding to the various states defined for the object. But the dashboard visual is actually nothing more than a image representation of the current state of a hardwarecomponent. Thus, the dashboard visual does not contain information or references to the model of the component it is visualizing. This is handled with a *value provider* to be described below.

DashboardInteractionVisual In this version of the dashboard editor, the DashboardInteractionVisual is identical to the DashboardVisual. This is due to early design discussions which led to the idea that a user should not be able to change states in an actuator which would then be represented by a DashboardVisual. But the user should be able to simulate change in the state of a sensor which should then be represented by the DashboardInteractionVisual. The actual implementation, however, differed from this approach and so the DashboardInteractionVisual is actually deprecated for now and is kept in the model to make sure the current implementation would not become unstable if the class was removed from the model.

BusVisual These objects correspond to instances of busses in the simulation runtime. The BusVisual provides a graphical view of a current bus load for a BusRTInstance. The BusVisual is an extension of the DashboardVisual and is distinguished in its own class due to its very specific function.

ValueProvider For each DashboardVisual to be useful, it should associate with an instance of the type ValueProvider. The ValueProvider is a link between the visual object represented by the DashboardVisual and the model of the simulation. It maintains the value of an attribute corresponding to an attribute in the Simulator Runtime. An instance of the class StateValueProvider could, for instance, maintain an attribute mirroring an On/Off state attribute in an actuator in the Simulator. This setup allows components and busses in the Dashboard to always show up-to-date information. Instances of the class IntegerValueProvider are used by BusVisuals to get the number of messages waiting in a bus.

2.2.5 Sub-model relations

Figure 5 shows a view of all the previously described models and their interrelation. The deployment model is made from instances defined in the component definition. This relationship is illustrated by the associations between component- and port- definitions and instances.

The simulation model represents a runtime instance of the deployment and is therefore associated through objects concerning components, ports, connections and busses. An additional association to the component definition is needed to make the simulation aware of potential component states.

Finally, the dashboard model is associated with both the deployment and the simulation models. The deployment association is needed to setup a new dashboard instance and the simulation is needed during actual simulation.

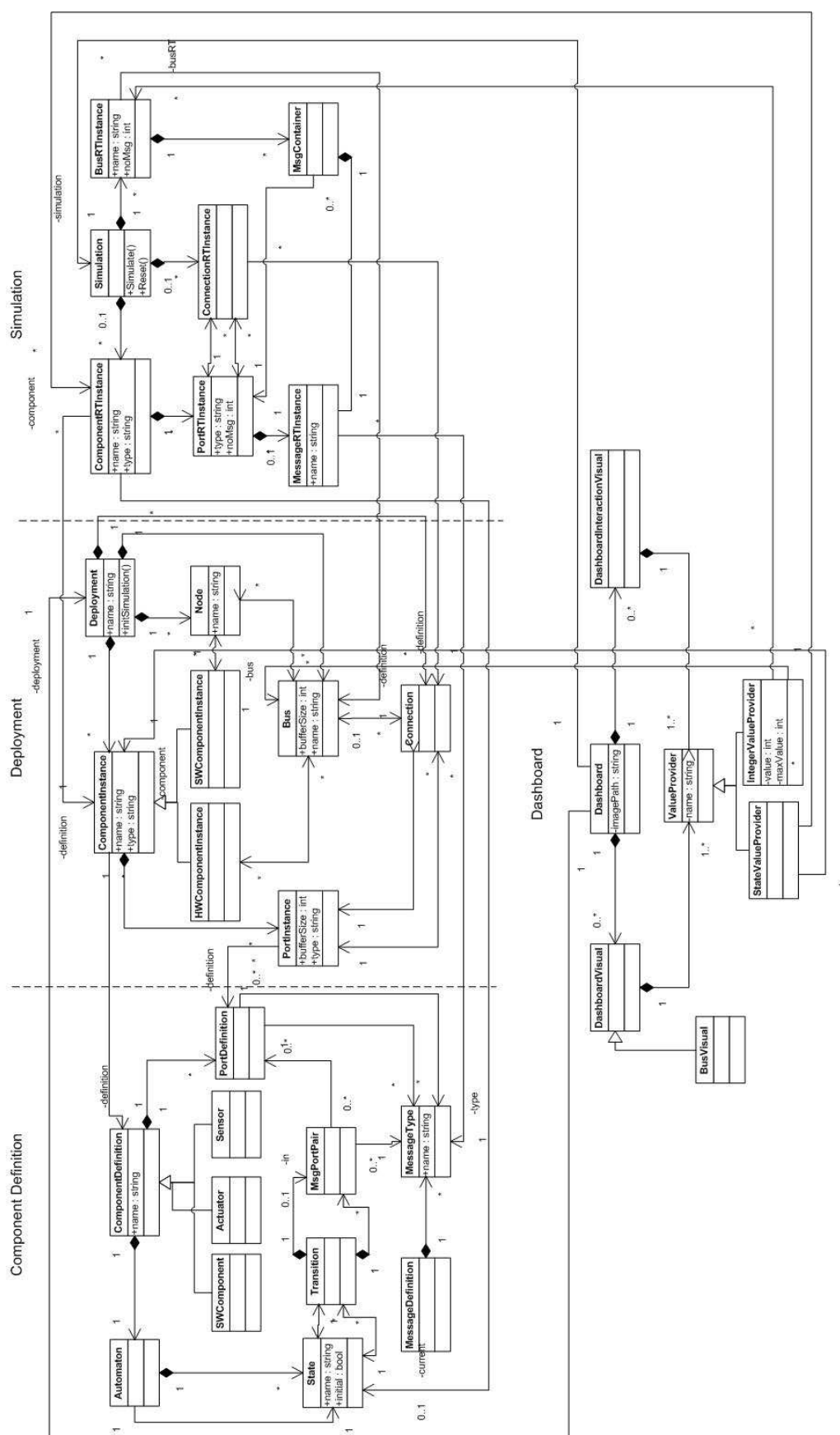


Figure 5: Class diagram for component, deployment and runtime

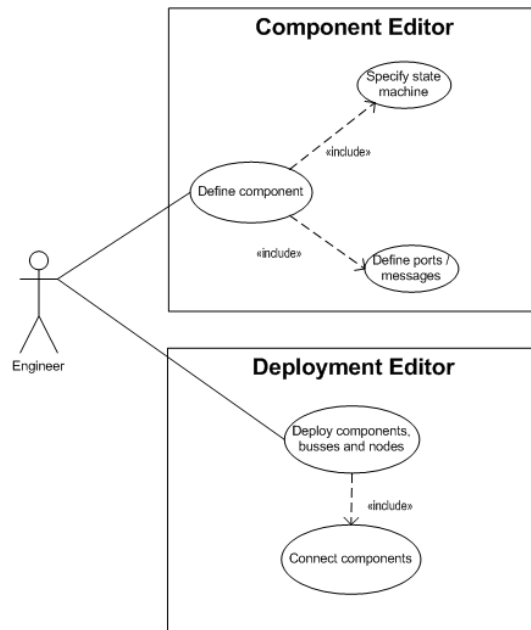


Figure 6: Use case diagram for component and deployment

2.3 Use Cases

The use case diagrams of the system provides us with an idea of how the system can be used and how it should react in certain situations. This subsection describes the various use cases for each part of the system.

We have combined the use case diagram for the component and deployment editor. This diagram is shown in Figure 6. The specific use cases are described in the following subsections. In a use case scenario, an underline means that we have an included use case which is described below the main succes scenario.

2.3.1 Component Editor

Define Component Primary Actor:

Engineer

Stakeholders and Interests:

Engineer wants to simulate an embedded system and needs to define some components for this.

Preconditions:

None.

Postconditions:

Main Success Scenario:

1. Engineer chooses component type(Controller/Sensor/Actuator)
2. Engineer gives the component a name
3. Engineer defines ports/messages
4. Engineer specifies the state machine
5. Engineer saves the component

Details:

1a. Engineer chooses Sensor or Actuator:

1. Engineer gives the component a name
2. Engineer defines ports/messages
3. Engineer specifies the state machine
4. Engineer saves the component

Special Requirements:

None.

Frequency of Occurrence:

Every time a system is initially set up.

Included use case: Define ports/messages Main Success Scenario:

1. Engineer chooses the number of ports and names them
2. Engineer writes in and out message list for every port

Included use case: Specify the state machine

Main Success Scenario:

1. Engineer creates and names a number of states and appoints an initial state
2. System registers the states
3. Engineer creates transitions between the states
4. Engineer assigns trigger and outgoing message pairs to the transitions

5. System ensures that the messages the ports can send and receive are compatible with the transitions trigger and outgoing message pairs
6. System registers the transitions and their trigger and outgoing message pairs.

Extensions:

- 4a. System discovers that the messages the ports can send and receive are not compatible with the transitions trigger and outgoing message pairs
 1. System warns Engineer about the inconsistency
 2. Engineer edits the ports/messages or the transitions trigger and outgoing message pairs
 3. Return to main Success scenario at step 4

2.3.2 Deployment Editor

Deploy components, busses and nodes **Primary Actor:**
Engineer

Stakeholders and Interests:

Engineer wants to model behaviour of the system.

Preconditions:

Engineer has defined a set of components, either from scratch in the component editor or by loading an existing list of components from a file.

Postconditions:

Engineer has made a hardware system using his own components, busses and connections. System is ready for simulation.

Main Success Scenario:

1. Engineer starts the deployment editor.
2. Engineer deploys a number of components, from his list of components.
3. Engineer deploys a number of busses and specifies their capacity.
4. Engineer connects the components.
5. Engineer saves his deployment configuration to a file.

Details:

*a. At any time, System fails:

1. System prints an error message to the engineer.
2. Engineer restarts the system and tries again.

2a. No components are available

1. System prints a message telling the engineer that he needs to define some components.
2. Engineer closes the deployment editor and starts the component editor.

2b. Engineer deploys a node

1. Engineer assigns a software component to the node (e.g. controller)

3-5a. Engineer wants to save his work and finish it some another time.

1. Engineer clicks the "save" button.
2. System saves his deployment.

5a. Engineer connects two incompatible components.

1. System prints an error message to the engineer.
2. The illegal connection is removed by the system.

Special Requirements:

None.

Frequency of Occurrence:

At least once for each new hardware system.

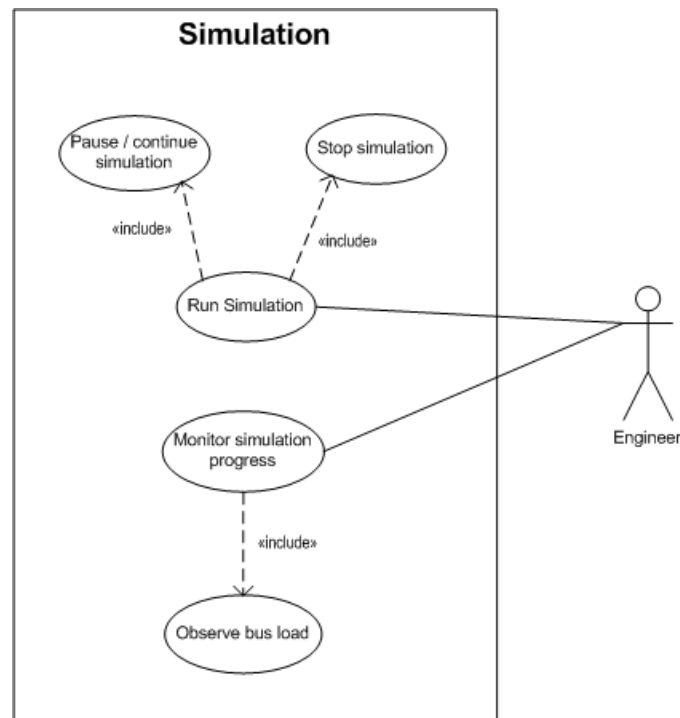
Included use case: Connect components

Main Success scenario

1. Engineer specifies which components to connect
2. A transition is made between the components

1a. The components are not in the same node

1. Each component has to connect to the bus



2.3.3 Simulation

Run simulation Primary Actor:

Engineer

Stakeholders and Interests:

Engineer wants to know if the system works correctly

Preconditions:

Engineer has made a deployment model.

Postconditions:

None.

Main Success scenario

1. Engineer or end user loads a deployment configuration from a file
2. Engineer specifies which components should start in another state than the initial.
3. Engineer starts simulation

4. System runs through an iteration of the simulation loop (See Simulator algorithm subsection 3.1)
5. System repeats step 4 until nothing new happens. E.g. no messages are to be sent between components.
6. System stops simulation algorithm and print out the results to the engineer.

Special Requirements:

None.

Frequency of Occurrence:

Every time an engineer or end user needs to simulate the system, eg. after every change in a deployment setup.

Included use case: Run a single step

During the simulation, the engineer might want to go through the simulation step by step, i.e. control the simulation manually.

Main Success scenario

1. Engineer clicks the single step button.
2. The simulator performs one step and then pauses.

Included use case: Pause and continue simulation

During the simulation, the engineer might want to carefully examine a certain state of the system. He therefore needs to pause the simulation.

Main Success scenario

1. Some time during the simulation, the engineer pauses simulation.
2. Engineer observes the current state of the system.
3. Engineer continues simulation.

Included use case: Stop simulation

At all time during the simulation, the engineer should be able to stop and exit the simulation.

Main Success scenario

1. System is running the simulation.
2. Engineer stops the simulation.
3. System stops the simulation and print out the result to the engineer.

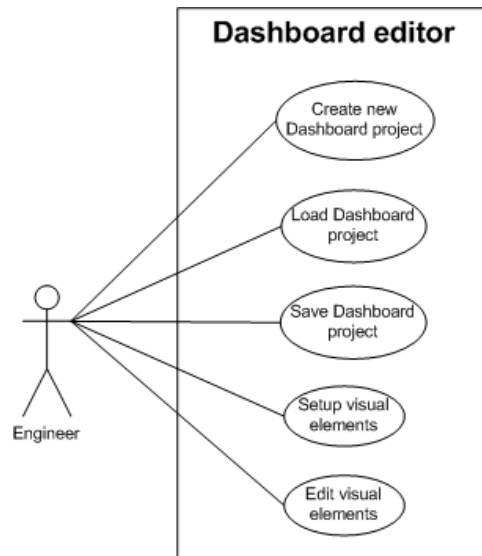


Figure 7: Use Case diagram for the Dashboard Editor

2.3.4 Dashboard Editor

Deployment/Dashboard Presentation

Create new Dashboard project **Primary Actor:**
Engineer

Stakeholders and Interests:
The engineer wishes to visualize a systems behaviour.

Preconditions:
Deployment exists.

Postconditions:
An empty Dashboard project has been created.

Main success scenario:

1. The engineer requests creation of a new Dashboard project.
2. The system queries the engineer for the location of the file which specifies the deployment model.
3. The engineer selects a deployment file.
4. The system responds by loading and displaying a lists of components.

Exceptions:

4a. An error is encountered while loading deployment diagram.

1. System warns engineer and prompts the engineer for retry or abort.

4b. The deployment file is invalid.

1. The system throws an error and closes the dialog.

Special Requirements:

None

Frequency of occurrence:

At least once for each deployment.

Load Dashboard project Primary Actor:

Engineer

Stakeholders and Interests:

The engineer wishes to continue work on a previously saved project.

Preconditions:

A Dashboard project has been saved to file.

3 Postconditions:

The Dashboard project is loaded and displayed in the editor.

Main success scenario:

1. The engineer requests load of existing Dashboard.
2. The system queries the engineer for the location of the file which specifies the Dashboard model.
3. The engineer selects a Dashboard file.
4. The system responds by loading and displaying the Dashboard model in the editor.

Exceptions:

4a. The Dashboard model file is invalid.

1. The system throws an error and closes the dialog.

Special Requirements:

None.

Frequency of occurrence:

Multiple occurrences for each deployment model.

Save Dashboard project Primary Actor:
Engineer

Stakeholders and Interests:

The engineer wishes to save his work in order to backup his progress or because he is planning to continue his work at a later time.

Preconditions:

A Dashboard model is loaded in the editor.

Postconditions:

The Dashboard model has been saved to file.

Main success scenario:

1. While a project is open the engineer requests a save.
2. The system queries for a file location used for saving.
3. The system saves the model to file.

Exceptions:

- 4a. The specified file already exists.
 1. The system prompts for override.

Special Requirements:

None.

Frequency of occurrence:

Multiple occurrences for each Dashboard model.

2.3.5 Dashboard

Changing component state Primary Actor:
End User or Engineer

Stakeholders and Interests:

The *end user* wants to see a simulation of what is happening, when the component changes state.

The *engineer* wants to see that everything works as it should, when a component changes state.

Preconditions:

The Dashboard has loaded the data provided from the Dashboard Editor and is connected to a Simulator

Postconditions:

The components state has changed which will then change the state of the entire system.

Main Success Scenario:

1. Actor selects simulation view
2. The user selects a component in the drop down menu
3. The user selects a new state for the component
4. System registers the new state.
5. The state of the other components will change.

Special Requirements:

None.

Technology and Data Variations List

None.

Frequency of Occurrence:

Every time a user wishes to change a state.

Viewing interactions Primary Actor:

Engineer

Stakeholders and Interests:

Engineer wants to see interactions between components, e.g. messages sent between components.

Preconditions:

The Dashboard has loaded the data provided from the Dashboard Editor and is connected to a Simulator.

Postconditions:

A view showing messages is visible.

Main Success Scenario:

1. Engineer clicks on “interactions” button.
2. A view showing interactions shows up.

Special Requirements:

None.

Technology and Data Variations List

None.

Frequency of Occurrence:

Every time the “interactions” button is clicked.

3 System Features for Simulator (s052271)

Whenever the engineer has made a deployment of components in the deployment editor, he can start simulating the system. A simulation is an instance of the deployment. For the simulation, the dashboard can be used. Here it is possible to trigger different components like sensors or actuators, which will start the simulator with their new state. The simulation can be run at a slow steady speed, or it can be run step by step. During the simulation it is possible to monitor the different components and their status.

3.1 Revised Simulation algorithm

For each iteration of the Simulator loop

 For each ComponentRuntime

 For each PortRuntime on ComponentRuntime.

 If there are messages on the Buffer

 Get first message from Buffer

 Follow Transition to new State according to message

 If new State has outgoing Message

 Add Messages to BusRuntime or PortRuntime

 For each ConnectionRuntime

 For each outgoing connection on source Port

 If port has outgoing messages on buffer

 If target Port is Bus

 Put first message on BusRuntime's Buffer

 Else (target is component)

 Put first message on components incoming Buffer

For each BusRuntime:

 Transmit one Message and update list of Messages.

4 System Features for Dashboard (s973748, s052233)

4.1 Create new Dashboard Project

Since a Dashboard is a visual representation of a deployment it must, for a Dashboard project, be possible to specify what deployment model to use. Deployment models are specified in XML files following an agreed upon schema. The first step in creating a new Dashboard project must then be to name the XML file which defines the deployment model instance.

4.2 Load Dashboard Project

Since Dashboard project as a rule won't be finished in a single session it must be possible to load previous project from a file. Like deployment models, Dashboard models are defined using XML and the primary action by the engineer when loading a new project will be to specify the relevant file.

4.3 Save Dashboard Project

As mentioned earlier, Dashboard project can be quite complex and it would be unrealistic to assume that a new project is created in each session. It should therefore be possible to save Dashboard models to file, either to use in the actual Dashboard or to continue work upon in the editor.

4.4 Visual Element Setup

Since the primary purpose of the Dashboard is to visualize component states, it follows that the primary purpose of the editor is to setup the parameters controlling these visualizations. Each new visual must be connected to a deployment component instance through a ValueProvider.

4.5 Visual Element Editing

In case of changes to the deployment model or planned Dashboard model it must be possible to change existing visuals. This will work by selecting an existing visual element and selected Edit. During the edit phase it must be possible to select another ValueProvider.

4.6 Sensors

4.6.1 Description

Sensors are the part of the system, the end user can interact with. This could for instance be a rain sensor on a car.

4.6.2 Stimulus/Response sequences

Stimulus A user changes the state of a sensor.

Response The sensor will go into the new state, specified by the user. In this new state, it will send a message providing information about what it has sensed.

The *end user* will be able to see this, when the actuator starts its animation.

The *engineer* will also be able to see other things, something we will go into details with later.

4.7 Actuators

4.7.1 Description

The *actuator* is the part of the system, that does some kind of action. This could for instance be a wiper for a car. Actuators will start their actions, once they receive a message which tells them to do so. This means that they are only active when some action occurs elsewhere. This would typically be a sensor sensing something, but it could also be a controller, that just tells the actuator to start.

4.7.2 Stimulus/Response sequences

Stimulus The actuator receives a message.

Response (*End user*)An animation will start, showing that the actuator has now started.

(*Engineer*)This will lead to the same response, but in addition, he will get information about the bus load.

4.8 Controllers

4.8.1 Description

End user A *controller* can receive and send messages, as well as do well-defined behaviors in a state machine. The controller won't be visible for the end user, since it isn't providing him with any useful information.

Engineer The engineer will also need to see where the messages are sent. However, to be able to identify possible problems in the deployment, he must be able to see what is happening inside the controller, i.e. in the state machine (see next section).

4.8.2 Stimulus/Response sequences

Stimulus (*End user*) A sensor senses something → sends a message to the connected controller.

Response The controller will receive the message and in this way get activated. The controller will then communicate with some other component, typically an actuator.

Stimulus (*Engineer*) A sensor senses something → sends a message to the connected controller.

Response Identical to previous response and in addition, since he can see the controller, he is able to see which port is used.

5 External Interface Requirement (s973748, s052271, s052327)

This section provides a list of the interfaces needed, to use the CASE tool.

5.1 User Interfaces

The interfaces provided by GEF and GMF.

5.2 Hardware Interfaces

None.

5.3 Software Interfaces

5.3.1 Simulator

SI-1 The deployment editor.

SI-1.1 It is assumed that the deployment editor exports specifications in an XML format. The Simulator must be compatible with this format.

5.3.2 Dashboard Editor

SI-1 The deployment editor.

SI-1.1 It is assumed that the deployment editor exports specifications in an XML format. The Dashboard must be compatible with this format.

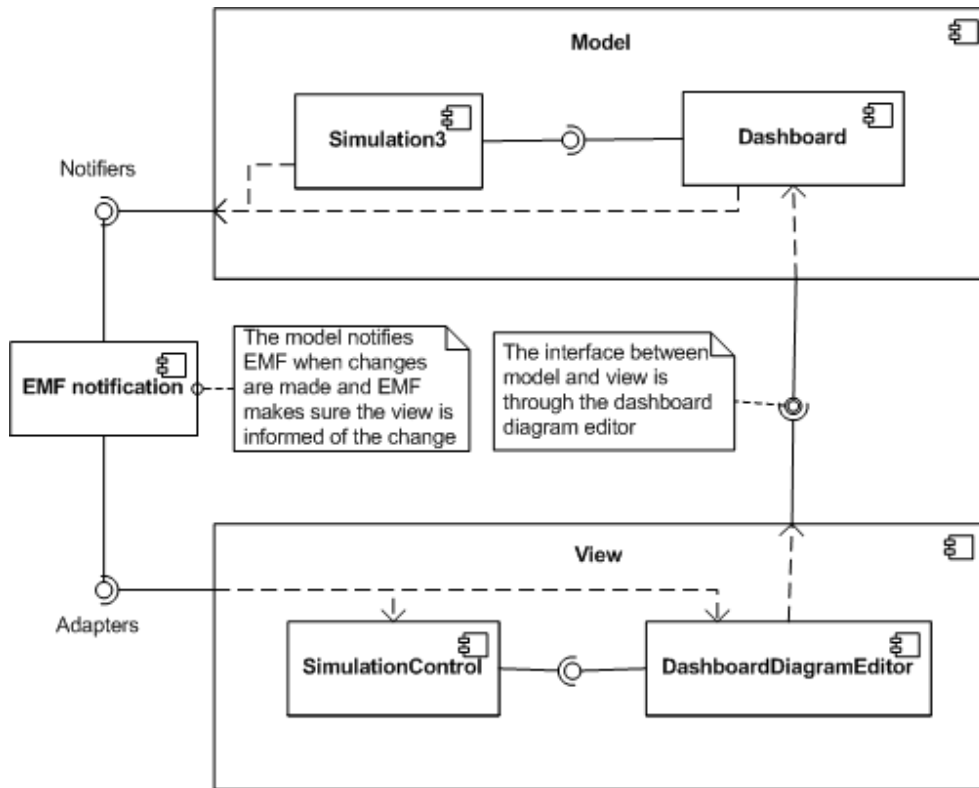


Figure 8: Interface between the model and the view.

6 Architecture(s052275)

The architecture of the dashboard editor is for the most part governed by the modeling framework it is built upon, ie. EMF and GMF. The core functionality for handling *visual* objects in the editor is defined through GMF and conforms to changes in the *model* of the dashboard which is defined in EMF. The editor also gives the user a way of *controlling* the model through interaction with the view and thereby controlling the view. The control can also be handed over to different participants such as a simulator.

From the brief description above we have actually identified which concept the architecture of the dashboard editor is primarily based on, namely the *Model-View-Control* (MVC). This concept is standard for EMF and GMF and we will not go into details and defining the concept, but it is important to be aware of the three parts of the program when describing the architecture because they are separate concepts but not always designed and implemented to be separate. The following sections will describe each part and point out the relevant interfaces between them. An outline of the interfaces are shown in Figure 8 which we will reference further when it is relevant.

6.1 Model

The dashboard editor is by nature a very graphical experience with the primary purpose of visualising a simulation of an embedded system. This fact is also the reason why the model of the dashboard editor is not very complex as seen in the UML class diagrams in earlier sections. What is interesting in this section is how the model interfaces between the different components such as the dashboard, the deployment and the simulation.

As described in the UML class diagram for the dashboard, the dashboard has references to both a simulation and a deployment. However, the simulation and the deployment do not have any connection to the dashboard. This does not matter because the dashboard is just a visual representation of the deployment and the simulation and they do not need to change the state of their own visual appearance - this would actually not make sense at all.

The interfaces of the model are setup and managed by EMF on the basis of the UML model of the dashboard editor, and they follow the rules and standards of EMF and will not be dealt with further.

6.2 View

The view is also maintained through GMF and GEF via auto generated code. The view consists of the actual dashboard diagram editor and a simulation control view which has been manually added. The view is updated by the built-in notification framework of EMF. The dashboard diagram editor is registered as an *adapter* to its model counterpart which notifies the dashboard diagram editor of changes in the model. This feature is shown in Figure 8.

The simulation control view is not an adapter to the model and is thus not notified of changes in the model. But the purpose of the simulation control view is also just to function as a one way communicator from the user to the simulation.

6.3 Control

The dashboard can be controlled by two objects: A user and the simulator. From the user perspective the control is done through the view itself, ie. the dashboard editor. This is possible through the use of EMF and GMF which automatically sets up the correct interfaces and makes it possible for the user to interact and control the model and the view. This control is restricted to adding, setting and changing objects and attributes for these objects in the dashboard diagram editor. Through the use of the simulation control view it is also possible for the user to control the simulation. This involves for example creating and starting the simulation. Since this view is manually added, an interface has been setup from the simulation view through the dashboard diagram editor and to the model. This is shown in Figure 8.

The simulation also controls program executing. While the user controls the model through the view, the simulation is part of the model itself and does not need an interface to control model changes. This makes control for the simulator straightforward and all the simulator has to do is to notify the view of changes made in the model.

The next section will go into more technical detail about the actual implementation of the architecture and also give more examples of how the model, view and control are connected and behave through the use of sequence diagrams.

7 Implementation and test(s052271,s052275,s052327)

7.1 Editor

7.1.1 Autogeneration of code

A large part of the dashboard editor has been implemented by autogenerated code. This is possible thanks to EMF and GMF. Although it is not a trivial and straightforward process to use EMF and GMF, it significantly speeds up the implementation during the early stage of development. This is partly because it removes the necessity of writing a lot of “simple” code. And with GMF it is fairly easy to setup a visual representation of a model without spending too much time on coding but rather thinking about modeling. The procedure we have used to implement the dashboard editor is summarised below:

1. First we create an Ecore model of the dashboard corresponding to the UML class diagram presented in an earlier section on Figure 4.
2. From the Ecore model we can create a generator model automatically. This generator model is able to autogenerate the model code for dashboard.
3. Three GMF files make up the definitions necessary for auto generating the dashboard diagram. These are:
 - The GMFTool which defines what tools should be available in the editor. This is for example a tool for creating a Dashboard Visual.
 - The GMFGraph which defines how graphical components appear in the editor. The Dashboard Visual could for example be defined to appear as a rectangle.
 - The GMFMap which maps the model to specific behaviour in the diagram. The Dashboard Visual is for example defined as a top node with a compartment consisting of State Value Providers.
4. From the above three GMF files, a GMF generator model can be created. This generator model autogenerates diagram code.

When using EMF and GMF the model of the dashboard (Figure 4) is quickly prototyped but still needs some extra touches as described in 7.1.2.

7.1.2 Implementation issues

EMF and GMF brought us some part of the way in implementing the dashboard, but we still had to make some additions to the generated code, especially in the Diagram Project.

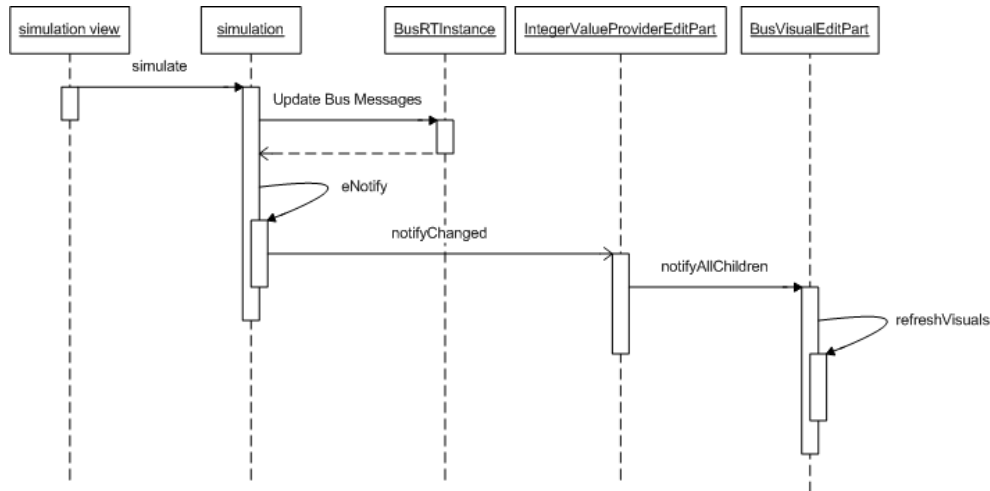


Figure 9: Sequence diagram showing a simplified notification chain.

Dashboard Diagram Project This section describes some of the changes we have made to the generated code. GMF doesn't provide much support when you want to use external images in your figures. Therefore we have made it ourselves, using our knowledge of GEF. Furthermore we have made a link between our simulator and the dashboard, so that the figures can be properly updated whenever the simulator is running. This is discussed in the architecture section

Dashboard Diagram Project This section describes some of the changes we have made to the generated code. GMF doesn't provide much support when you want to use external images in your figures. Therefore we have made it ourselves, using our knowledge of GEF. Furthermore we have made a link between our simulator and the dashboard, so that the figures can be properly updated whenever the simulator is running. This is discussed in the architecture section.

Every time the simulation has executed a single step, the whole dashboard is notified about it. Each *Editpart* class has a method called `notifyChanged()`, which we have overridden. Figure 9 gives a simplified example of how the notification is actually performed. Each of the Editpart classes react somewhat different from each other. Here follows a detailed explanation:

BusVisualEditPart This class represent the graphical representation of the BusVisual object. It is used to depict the current load on a specific bus via a *IntegerValueProvider*. The Busvisual figure has a standard look, that cannot be changed by the user. The current load is visualized in a bar that changes colour depending on how big the load is. The BusVisual

figure is designed using only standard Draw2d components. Whenever the this class is notified about a change from the simulation, it will get the new load information from the model, calculate a new size and colour for the bar and paint it.

DashboardVisualEditPart This class represent the graphical representation of the DashboardVisual object. It uses external images to depict the current state of the ComponentRuntime it is associated with. Whenever the current state is changed, the image should be changed as well. This is done in the method `refreshVisuals()`. The *Image path* property which is specified for the dashboard, is used to look for image filenames which corresponds to the current state name. If such an image exists, it will be loaded and shown on screen. If there is not an image available or if the state name is empty, a default image will be loaded. The default image is specified by putting an image in the image directory, renaming it to “default.X”, where X is a standard image file extension (e.g. GIF, PNG, JPG etc.).

IntegerValueProviderEditPart This class provides some number data to be visualized in a figure. It is primarily used in the BusVisual. We have made this class an adaptor for the dashboard. It means that it will be notified whenever the dashboard is notified.

Whenever this class receives a notification, it will make sure that its sibling (the graphical part of the figure) gets the notification as well. This valueprovider is visualized as a small image icon. This image icon should be installed as described in our user manual.

StateValueProvider2Editpart This class provides state data to be visualized in a DashboardVisual figure. We have made this class an adaptor for the dashboard. It means that it will be notified whenever the dashboard is notified. This valueprovider is visualized as a small text label positioned in the top of the DashboardVisual. When a ComponentRT is associated with this valueprovider, the label will show the component name and current state.

Whenever this class receives a notification, it will update the text label. It will also make sure that the DashboardVisual gets the notification as well.

SimulationControl Our simulation view is implemented in the class `SimulationControl` which is in the package `dashboard.diagrams.views`. As described in the subsection about the simulator, this class is purely graphical. It does not do much else than calling relevant methods in the simulator.

We had a hard time figuring out how to connect this view with the simulation in the dashboard. Our solution is perhaps not the perfect way to do it, but it works: The view implements a so-called `IPartListener`.

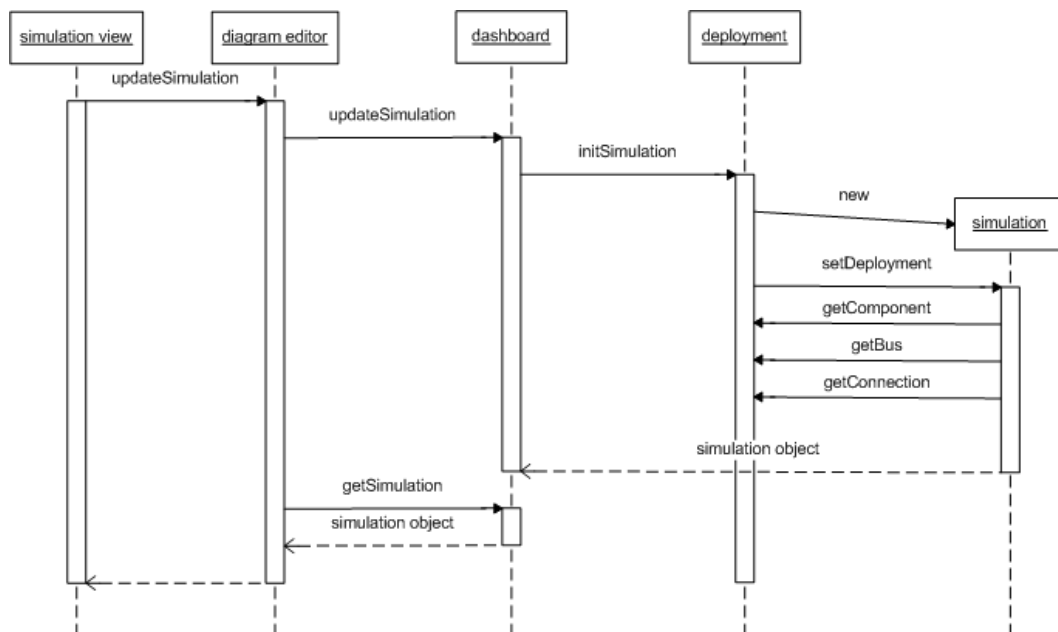


Figure 10: Sequence diagram showing how the simulation view creates a new simulation

It listens for changes in all views in Eclipse. So whenever a dashboard is opened or closed, we are able to hook up or unhook our simulation view with this dashboard. The listener isn't perfect, sometimes it doesn't react on a dashboard being opened. If this is the case, the user should manually hook up the simulation view with the dashboard (See user manual).

DashboardDiagramEditor The actual editor for the dashboard diagram is implemented automatically by GMF in the class `DashboardDiagramEditor`. This class functions as the link between the simulation view and the dashboard model. The class had been added with two important methods. A `getSimulation` method which gives the current instance of the simulation object from the model. And a `updateSimulation` method for updating the simulation object.

The latter method calls the `updateSimulation` method in the dashboard model and thus creates a simulation object for running a simulation of the dashboard. The simplified sequence diagram in Figure 10 shows the program flow when calling the method from the simulation view. The method also makes sure that any visual objects on the dashboard, which has been associated with a value provider, also gets associated with the proper runtime instance from the simulation object. For this to work a model change has to be made and updating of the model is done through a *transactional*

editing domain which is the only way of changing the model from different threads. This domain is only accessible (or rather: *easily* accessible) through the dashboard diagram and not the model which gives the reason why this logic has been put in the diagram editor and not the model itself.

7.1.3 Test

No indepth test has been made of the dashboard editor. This is partly because the dashboard editor is based on EMF and GMF and we can only expect that the developers of these technologies have tested the integrity and functionality that the autogenerated code provide.

Testing the dashboard editor has primarily been functional testing. We have tried out the features of the dashboard, setting up different settings and seeing if the simulation view worked properly. There have not been a fully systematic approach but our effort has mainly gone into testing the simulator as described in the next section.

7.2 Simulator

7.2.1 Simulation

The CASE tool enables a user to create components, deploy them and put them into a dashboard. The dashboard should then show the user what will happen, when the system is running. To do this, we need to *simulate* the system. As we have seen, the system consists of components and busses connected via user-defined `Connections` in the deployment. Every port on a component and each bus has a buffer which can hold a number of messages. Sending messages between the components is then done by putting messages on these buffers.

To simulate this behaviour, we need to define some rules for the simulation. First of all, the system we are simulating is an embedded system. This is an important fact, since there will be limited resources, so we can't just let the components send all messages from a buffer at once. In order to have full control of the simulation, we need to do the simulation in *steps*.

Simulating a step We must decide how much work each component and bus should do in a single step. To decide this we must look at how a component is built. A component consists of:

- An automaton/state machine
- One or more ports

It is not realistic that the entire buffer should be processed in a step, but still we must choose whether all enabled ports¹ should be processed in each

¹An enabled port is a port without a trigger or a port with a pending ingoing message.

step or if only one should. By choosing to process only one, we could risk that some messages may never be sent unless there is some sort of queuing. However, this is a very small risk, since mostly when sending a message, the component will go into a new state, where the same message cannot be send again (and be put on the buffer again). Since a step usually means doing exactly one thing, we've decided to process only a message on one enabled port in each step.

By choosing this solution, we also choose that it is only possible to fire one transition in a component per step, since it requires a message to fire a transition.

The problem with the busses is analogous to the components, but now we have exactly one buffer (instead of one for every port). By implementing the buffer as a FIFO-queue² we assure that every message eventually will be processed.

As with the components, we've decided to process one message in every bus in each step.

7.2.2 Algorithm

The previous subsection described the simulation – what it is and how it is supposed to work. In this subsection we will go into details with the actual implementation of the choices we have made.

Creating the simulation The model of the CASE tool describes how the component definitions, the deployment and the simulation are related. Each component has a definition, an instance and a runtime instance. In the component definition we created the definition of the component. In the deployment this definition was instantiated. To simulate the instance of a component, we need a runtime instance. Since this isn't created automatically, we must implement a method which given a deployment creates runtime instances of each component, bus, port and connection.

For each component we must create a runtime instance, which knows its definition, its simulation and its current state in the automaton. Every port in the component also needs a runtime instance. Since it needs to know what component it belongs to this is done during the creation of component runtime instances.

Likewise, for each bus and connection, we create a runtime instance, which knows its definition and simulation, so that we can use it when simulating.

A single step In subsection 7.2.1 we have discussed various ways of simulating a single step of the communication in an embedded system. Now

²First In First Out

```

1  For Each ComponentRTInstance Do
2      For Each PortRTInstance Do
3          If the buffer has messages Then
4             
5              Take the first message from buffer
6              Find transition this message triggers
7             
8              If no such transition Then
9                  Discard message
10             End If
11            
12             Fire transition
13             Get outgoing messages from transition
14            
15             If connected to bus Then
16                 Put messages on bus
17             Else
18                 Put messages directly on components
19             End If
20         Else
21             If the current state has transition with no triggers Then
22                
23                 Fire transition
24                 Get outgoing messages from transition
25                
26                 If connected to bus Then
27                     Put messages on bus
28                 Else
29                     Put messages directly on components
30                 End If
31             End If
32         End If
33     End For
34 End For
35
36 For Each BusRTInstance
37     If the buffer has messages Then
38         Take the first message from buffer
39         Put message on destination components port
40     End If
41 End For

```

Figure 11: Pseudo code for the simulation algorithm.

we know *what* to implement, but we still need to find out *how*. We need to simulate the behaviour of several *individual* components. We emphasize that they can be seen as individuals, since the implementation of sending and receiving messages for one component is independent of the other components. This means that two components could actually read messages on their buffers *at the same time*. This gives us the possibility to implement the simulation as a multi-threaded application. On the other hand, by implementing the simulation multi-threaded we face some aspects of concurrent programming which could give problems. What would for instance happen if two components tried to put a message on the same buffer at the same time? Because of these issues which could make the simulation a lot more complex than needed, we've chosen not to implement a multi-threaded simulation. Instead our implementation is based on iterative programming. This way we make sure that only one component will access a buffer at a time.

In each iteration a new component is considered. In this component we iterate over the ports to find one with a buffer that isn't empty. This leads to two different outcomes:

- **A buffer with a message is found**

If this is the case, the first message from the buffer is taken and examined. If there exists a transition where this message is a trigger, then the message is valid and the transition should be fired. Firing a transition means sending all outgoing messages on this transition and changing the components state to the target state. When this is done, the algorithm is done examining this particular component and will continue to the next. If there are no transitions with this message as a trigger, the message is invalid and should be discarded.

- **All buffers are empty**

There can be two reasons for this: Either no one has sent a message, so we should just go exit this component and examine the next one. The other possibility is that the current state has no trigger messages. This means that no component can send a message with triggers this particular transition. If this is the case the transition should be fired anyway and any outgoing messages should be sent. *This is only done if all buffers are empty!*

This is basically what is happening in each component in a single simulation step. After this, all the busses are considered. In each bus we only need to check whether there is a message on the buffer. If this is the case, the first message is taken and sent to the correct receiver. If the buffer is empty nothing will happen in the particular bus. When all busses and components have been considered the simulation step is done.

See figure 11 for the pseudo code for the algorithm.

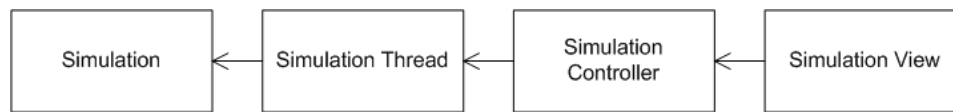


Figure 12: The structure of the simulation. The implementation of the simulation steps is in the leftmost box. The arrows indicates that the owner of the arrow controls the box at the end of the arrow.

Controlling the simulation In a larger embedded system it could easily become a tedious task to simulate behaviours using only the simulation algorithm we’ve discussed in the previous subsection. Sending a message from one part of the system to another could take many steps, meaning that one would have to do alot of steps before seeing any results at all. For this reason we need to be able to *control* the simulation by allowing to do several steps without clicking a step button each time.

This could be done in different ways: For instance, we could allow the user to specify a number of steps to perform when the step button is clicked. Another solution is to have a “play” button, which performs steps continously with a small pause between each step. Since it is not always known how many steps one would like to perform, we’ve chosen to implement a play button.

This gives us the following possibilities for controlling the simulation:

Step Perform a single step.

Play Simulate continously.

Pause Temporarily stop the simulation.

Stop Stop and reset the simulation.

Interval Choose how often a step should be performed when playing.

To be able to continously perform simulation steps and pause this, we need to implement a *Simulation Thread*, which will use the java function `Thread.sleep()` to maintain the interval between each simulation step when simulating continously.

The simulation thread isn’t able to control the simulation per default. To control the simulation thread and thereby control the simulation we have created a *Simulation Controller*, which uses methods from the simulation thread to control it. The methods used to put values into variables and notify condition queues in the thread.

To use the simulation in the dashboard we have created a *Simulation View* which consists of buttons using the methods from the simulation controller. Of course all the methods in the simulation controller could be implemented directly in the view, but by doing it this way we are actually using the Model-View-Controller design pattern, as described in section 6.

It also allows us to use the simulation without the dashboard, since it can be controlled without a view.

Figure 12 shows the relations of each part of the simulation. As we can see, each part is only controlling the next part. This also means that we could actually use a different simulation with the same controller and thread, as long as it is implemented using the interface provided!

Resetting the simulation It should be possible to reset the simulation, i.e. setting every component back to its initial state etc. For this purpose, we have implemented a `reset()` method, which clears the buffer of every port and bus and sets every component back to its initial state.

Since only states and buffers can change in the simulation, this is sufficient for resetting the simulation.

7.2.3 Test

In this subsection, we will describe the test phase of the implementation of the simulation. We have decided to test some of the simulators most simple functions, i.e. sending messages between components and busses. For this, a simple deployment has been created.

We could perform the tests on larger systems, but since the implementation of the test cases is tedious to make, and will be quite complex on larger systems, we have chosen to test it on a very simple system. By doing this, we ensure that the simulators ability to send messages between components work as expected.

It should actually be sufficient to test on a small system since we could just see a large system as many small systems put together. It also means that if we really want to test a larger system, we could just use a tool like the dashboard. This enables us to see that the components are reacting as they should.

Test setup Figure 13 shows the test setup we are using. We have created a very small deployment for testing purposes, consisting only of one software component, `CTRL`, one hardware component, `comp1` and a bus. The hardware component can only receive messages and the software component can only send messages. Each component consists of one port and a small automaton.

As we can see, the automaton in `CTRL` only has two states and one transition going from state one to state two. This transition has no trigger message, thus it is executed whenever the component is in state one (See subsection 7.2.2). Since the initial state is state one, the transition should be fired whenever the simulator is simulating the component the first time.

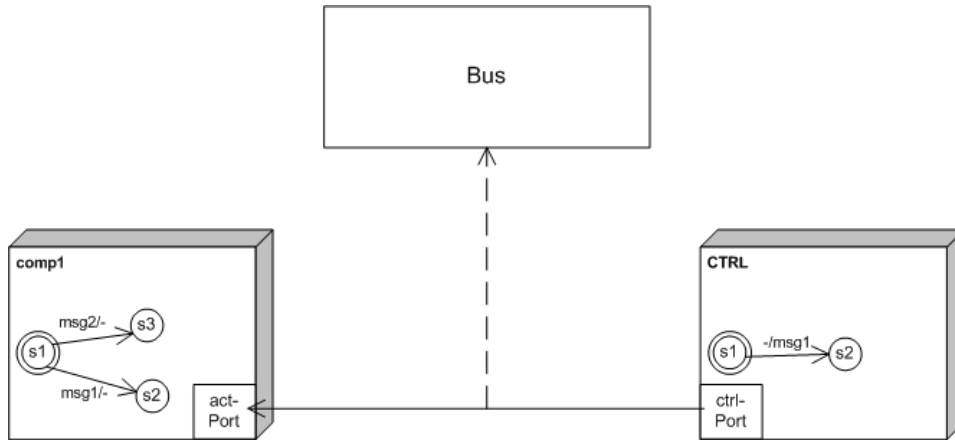


Figure 13: The figure is showing the setup we are using for the tests. It is a very simple setup, consisting only of two components and a bus. The dashed arrow indicates that the components communicate via the bus.

Performing the tests We want to test whether the simulation is able to send correct messages between components and busses. The deployment we have created only enables us to perform one test: Sending a message from one component to another. This is because CTRL has a transition without a trigger, meaning that this is send when the simulation is started. By starting the simulation, we actually test if sending messages between components work.

We want to start with an even more simple case though. When two components communicate, it is usually done through a bus (that is the case here). But how can we test this, if we do not know, whether a bus can send messages to a component? To perform a test like this, we need to place a correct message on the buffer of the bus.

This isn't quite as easy as it may sound. When then simulation is checking, whether a message is correct, it is comparing it's *Java object*, meaning that we cannot just create a new `MessageType` with a correct name: We need to find exactly the `MessageType` object which the component understands. Since every part of a simulation is located in lists, we need to loop through these to find the correct message. One could easily imagine that in larger systems, this would be very troublesome. By using the simple deployment, we have the advantage of for instance being able to pick out the ports easily, since there's only one port in each component!

Test cases The following gives a description of each test case: What it is testing and what is supposed to happen in each simulation step.

Test Case 1 The purpose of this test case is to test whether a bus is able to find messages on its buffer and send the first (oldest) message to the correct receiver. The messages are assumed to be valid. In this test, the bus will send the message `msg2` to `comp1`.

Expectations

As we can see in figure 13 the message is a valid message, so we would expect that `comp1` will change state from `s1` to `s3`. The simulation is done in steps so we cannot expect this to happen in the first simulation step. Instead we must analyze what will happen:

Step 1 The bus examines its buffer and finds `msg2` addressed to a port on `comp1`. The bus puts the message on the correct port.

Step 2 `comp1` checks the buffer on every port and finds the message from the bus. `comp1` finds a transition where this message is a trigger. `comp1` fires this transition and changes state to `s3`.

As we can see, two simulation steps should be enough for a component to receive a message from a bus.

Test Case 2 In this test we are testing whether a component can transmit a message to another component through a bus. From the previous test we know that sending messages from a bus to a component works, so this test is actually testing if a component can transmit a message correctly to a bus. This test will use the setup without any changes, since CTRL sends `msg1` to `comp1` when we are starting the simulation.

Expectations

Figure 13 again shows that this is a valid message and that `comp1` should change state to `s2` after some simulation steps. The number of steps expected for this test is again two:

Step 1 CTRL checks its buffer on every port and finds no messages. Since the buffer is empty, we now check whether the component has any transitions without a trigger. We find such transition and fires it. CTRL puts `msg1` on the bus' buffer.

The bus examines its buffer and finds `msg2` addressed to a port on `comp1`. The bus puts the message on the correct port.

Step 2 `comp1` checks the buffer on every port and finds the message from the bus. `comp1` finds a transition where this message is a trigger. `comp1` fires this transition and changes state to `s2`.

One might have thought that this test would need to simulate more than two steps, but this is due to the way we have chosen to implement the simulation algorithm. First every component is examined and *then* we check the busses. By that we can conclude that if a component puts a message

on the buffer of a bus, it will be found when examining the bus. This will happen in the same simulation step.

Test Case 3 The first two tests showed that sending messages between components through a bus works. We also want to know what happens when a component receives a message which is not valid for this particular component. In this test we send an invalid message `msg3` from the bus to `comp1`.

Expectations

As we can see, `msg3` is not a valid message for any of the components. When `comp1` receives the message, we would then expect it to discard the message. This will happen in two steps like before, but with some differences in step 2:

Step 2 `comp1` checks the buffer on every port and finds the message from the bus. Since it cannot find any transition with this message as a trigger, it concludes that it is an invalid message and discards it.

From this we can conclude that `comp1` will *not* change state, i.e. stay in state `s1`.

Test Case 4 Some components are not able to receive messages, i.e. they can only transmit messages. An example of such component is a sensor³. In our simple example, the component `CTRL` cannot receive any messages. We want to test what happens if a message is send to this component anyway. As described in subsection 7.2.1 we've decided to allow the message to be sent regardless and let the component discard it.

Expectations

We will expect that `CTRL` discards the message and stays in the same state. We know what will happen in step one, but step two is a bit different:

Step 2 `CTRL` checks the buffer on every port and finds the message from the bus. Obviously it will not find any transitions where this message is a trigger, since it has no triggers at all! It will then discard the message.

Since `CTRL` discards the message, we can conclude that it will not fire any transitions and remain in its current state.

Results We have implemented each test case as JUnit test cases, one JUnit case for each test case. This enables us to use JUnit's assertions to check that the resulting state is actually the expected state.

Running the JUnit tests went as expected: In each test the expected state was also the actual state. Of course, it is not possible to say that there

³Of course, some sensors will be able to receive messages that, for instance, tells it to stop sensing.

are no errors in the application, but these tests have showed us, that the basic functionality works as expected. Since most systems consists of components doing only these basic functions (sending and receiving messages) we can conclude that for the purpose of the simulation, these test cases are adequate.

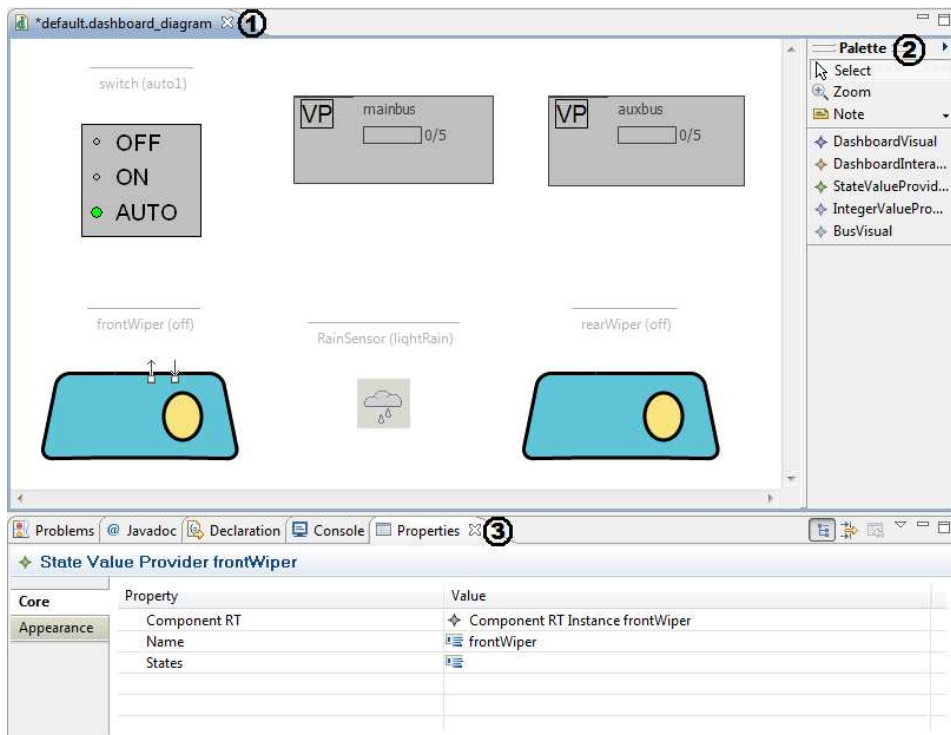


Figure 14: Screenshot of the Dashboard Editor

8 User Interface(s071150, s052233)

The purpose of this part of the document is to describe the Dashboard user interface. The entire user interface and all the functionality that can be achieved through it is described here.

8.1 Dashboard Editor

8.1.1 Loading a deployment into the Dashboard Editor

The Dashboard Editor will be creating a dashboard from a deployment. In the editor, the user is able to assign ValueProviders to each component. To do this, he needs to be able to load a deployment into the editor.

This is done by right clicking and then select the “Load Resource” option.

8.1.2 The main window(1)

In the middle of the main window you have the view window. The purpose of the view window is to show a graphical representation of the embedded system (deployment) developed using the deployment editor.

8.1.3 The palette(2)

The palette which is located to the right of the view window is a list of different tools the user can use to interact with the components in the view window.

The palette contains the following tools

- **Select**

If the user clicks on the select icon in the palette and then clicks on a component in the view window, that components information will be displayed in the properties window which is located to the left of the palette (the properties window is explained in section 8.1.4).

- **DashboardVisual**

This tool is used to create a figure which represents a component of the system. A DashboardVisual needs a StateValueProvider to be related to a ComponentRTInstance from the Simulation. Optionally, it is possible to define images for the different states to have a better overview of the behaviour of the component during the simulation.

- **DashboardInteractionVisual**

This tool must not be used. It is still in the editor to avoid unexpected errors if it is removed. **StateValueProvider**

This tool is used to provide the current state of a ComponentRTInstance from the Simulation. It must be created inside a DashboardVisual. **IntegerValueProvider**

This tool is used to provide the current load of a BusRTInstance from the Simulation. It must be created inside a BusVisual figure.

- **BusVisual**

It is used to create a figure which represents a bus. It needs an IntegerValueProvider to be related to a BusRTInstance from the Simulation.

8.1.4 Properties view(3)

Properties tab It is situated under the central panel by default and it is used to change the different properties of the selected element. If it is not showed, it can be opened from the menu Window/Show view and then selecting Properties. Following, the properties tab for each visual is described.

Dashboard It is showed when you click in any white area of the editor.

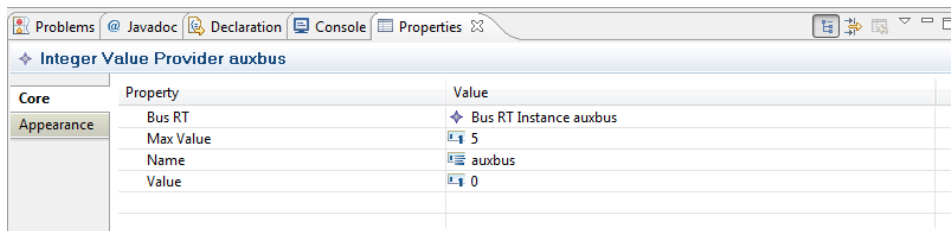


Figure 15: Screenshot of the properties tab for a bus

Property	Description
Deployment	It determines the deployment to be used. If the deployment does not appear in the selection list, it should be loaded. For that, do right-click on any white area of the editor and select Load Resource. Now, select the deployment file that you want to use.
Image Path	It points to the path where the images for the different states are located. The image whose file name (without the file extension) matches a state name will be visualized to represent that state in any component. The dashboard supports the most common image formats like GIF, JPEG, PNG, BMP, TIF.
Simulation	The simulation to be used. If the simulation does not appear in the selection list, you should create it in the Simulation tab.

Bus Visual It is showed when you click in a Bus Visual figure.

Property	Description
Value	It points to an IntegerValueProvider. If the value is empty, you should create an IntegerValueProvider inside the bus and this property will be updated automatically.

Integer Value Provider It is showed when you click in an IntegerValueProvider figure, which should be inside a BusVisual.

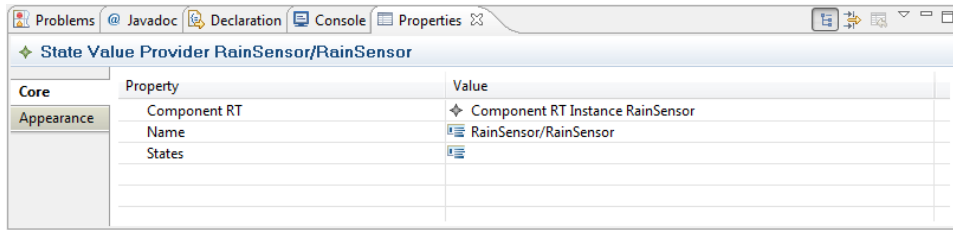


Figure 16: Screenshot of the properties tab for a Visual

Property	Description
Bus RT	It determines which Bus RT Instance should be used during the simulation. If none Bus RT Instance appears in the selection list, you should create the simulation in the Simulation tab. Then, if the deployment had at least one bus defined, you will be able to select some Bus RT Instance.
Max Value	It shows the capacity, in number of messages, of the Bus RT Instance selected in the Bus RT property. This property is updated automatically when the Bus RT property is configured.
Name	Name to identify the bus. The name is also updated automatically when the Bus RT property is configured.
Value	This property is the current load of the bus. It is not updated because it is already showed in the BusVisual figure.

Visual It is showed when you click in a DashboardVisual figure.

Property	Description
Value	It points to a StateValueProvider. If the value is empty, you should create a StateValueProvider inside the DashboardVisual figure and this property will be updated automatically.

State Value Provider It is showed when you click in a StateValueProvider figure. Specifically, you should click on the name “StateValueProvider” inside the DashboardVisual figure to have it selected.

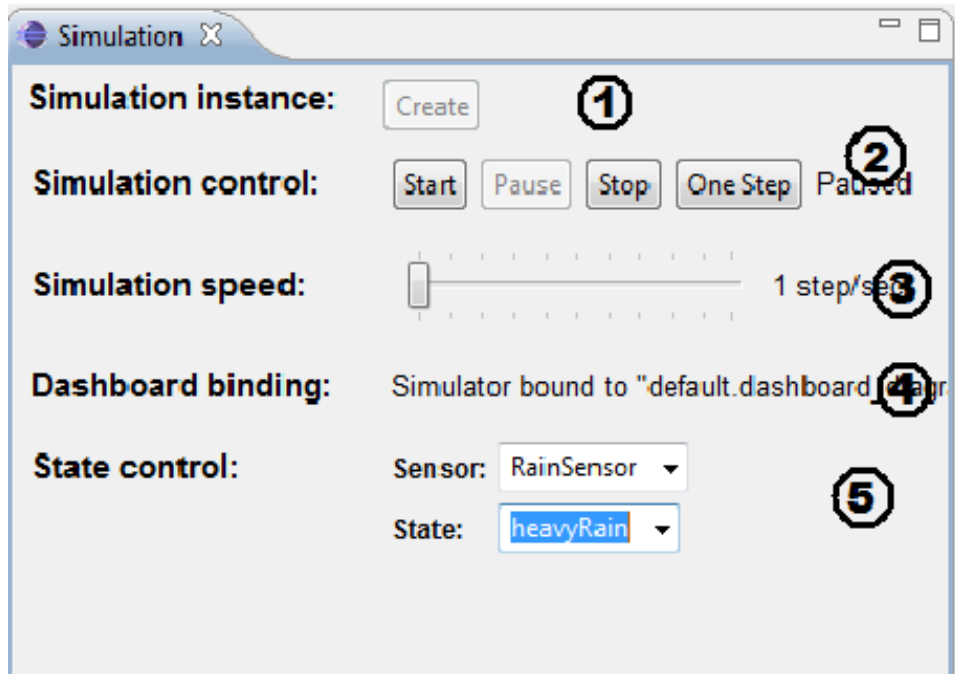


Figure 17: The simulation view

Property	Description
Component RT	It determines which Component RT Instance should be used during the simulation. If none Component RT Instance appears in the selection list, you should create the simulation in the Simulation tab. Then, if the deployment had at least one component defined, you will be able to select some Component RT Instance.
Name	Name of the Component. It is updated automatically when the Component RT property is configured.
States	This property is not used.

8.1.5 Simulation View

Simulation tab The simulation tab is used to start and control a simulation. If the simulation tab is not showed, it can be enabled by choosing “Window->Show View->Other->Dashboard->Simulation“

The view contains of five different parts

(1) Simulation instance

Is used to create a new simulation instance. It must be done every

time you want to simulate a new system.

- (2) Simulation control
Here you can start, stop or pause the simulation. Its also possible to do the simulation step by step.
- (3) Simulation speed
Controls the speed between each simulation step, when running the simulation
- (4) Dashboard binding
Shows which dashboard the simulation is bound to.
- (5) State control
Here its possible to change the states of the different components, by first choosing a component, and then select the new state.

9 Discussion and conclusion (s052275)

During this project we (group 3) have designed and implemented a dashboard editor/viewer as part of a computer aided software engineering tool (CASE tool). This report is the documentation for the work we have done during the project. The dashboard is now correctly implemented and working under the conditions of an environment described in this document and the user manual. This section discusses some issues we have had during the project and gives an outlook for future releases and finally a conclusion of the whole project.

9.1 Problems encountered

During the development of a large scale software project as the CASE tool, problems are discovered in the various forms. The first and most obvious problem we had was actually understanding what the project was about. This might seem strange but is just an example of how important it is to discuss and specify exactly what has to be done. The requirement specification and models which was written early in the project was very important in understanding the goal and pointing in the right direction. Even until the very end of development, a project is not necessarily understood by everyone and people might still have different opinions on how the task should be completed. However, we think that our end product does not suffer from this and actually, one of the goals for the project was to learn to handle this problem and we think we have accomplished that.

Another problem encountered was using the supplied technologies. Since modeling tools in general are not perfect yet, this fact also applies to the technologies used in this project. EMF and GMF have great potential but some parts are currently very bad documented, tutorials are hard to understand and the community is very minimalistic compared to major programming languages like Java or C#. This will hopefully be better as the technologies are maturing and more people start using them.

When building software on top of different technologies like EMF and GMF there will be a need for spending time on reverse engineering. In this project much of the implementation hours used was actually reverse engineering the structure of the EMF and GMF autogenerated code. The above problem with missing documentation made this a difficult and sometimes frustrating task. We have however managed to figure out how the important and necessary parts of the structure works so development had not been stopped because of reverse engineering problems which could easily have been the case.

9.2 Future releases

With the dashboard editor we have come a long way in visualizing and simulating an embedded system designed with the CASE tool. But the program could easily be upgraded and improved. One of the very important and also necessary improvements is to integrate all the different parts of the CASE tool. In this final version, the parts are still very separate although they can be used jointly by running them from different locations. It would be nice to have just one tool where all the parts are perfectly integrated and this would be an obvious task for future releases.

Concerning the dashboard there is still some open issues and improvements that could be made. The original idea was to have two different editors. One for the engineer to actually setup and edit how a simulation would run and a viewer for both the engineer and end user which should do nothing more than simulate the setup. Our solution was to merge the two and this works very nicely but maybe, future releases could *rethink* what possibilities there are in terms of giving an engineer and end user different experiences.

The visual appearance of the dashboard is fully functional and works very well in our version. This is of course provided that you specify an image path and have images with the correct names in this path. Although this works very well, there could possibly be a more intuitive and *user friendly* way of doing this. Another improvement is to include the possibility of having animated pictures such as GIFs which would further improve the usability and overview of the dashboard.

Differentiation between users using the dashboard is not implemented in our version. Every user sees the same output and has the same possibilities. In a future version the dashboard could perhaps (as also noted above) be split in two or more parts or the user could *authenticate* themselves when using the dashboard and thereby distinguishing them from one another.

9.3 Conclusion

The final result of the dashboard is very satisfactory and beyond what we ourselves had expected in the beginning of the project. The dashboard is correctly implemented and documented. The simulator is running smoothly and correct and combined with the dashboard, we are able to visualize and simulate an embedded system.

But the purpose of the project was not only to develop the CASE tool. The group project has also been a great learning process in team collaboration and communication. Every group member has benefitted from this and has achieved greater understanding and skills for handling larger software projects. The different phases of development, distribution of work and integration are important concepts for a software engineer and this project has helped us improve these attributes.

All in all we are satisfied with the result of the project. We think that we have met the objectives and created a functional and nice software program.